INITIALISATION

Use {} to initialize. An error will be thrown if an invalid type is used. Whereas () would try to use coercion:

int n = 6.23; // coerced to int 6

int n {6.23}; // error!

FUNCTIONS

Functions returning a pointer to an object allocated on the heap are dangerous. E.g.

Shape\* read\_shape();

Use smart pointers instead:

Unique\_ptr<Shape> read\_shape();

This will delete the object when not needed.

Function return types can also be written like (although not tradition):

auto name(arg) -> int {}

constexpr functions

Evaluate the initializer at compile time, e.g.

Constexpr int fac(int n) { return n; }

Int f5 = fac(5); // may be evaluated at compile time

Int fn = f(n); // error. Evaluated at runtime (n is a variable)

To be evaluated at compile time, a function must be suitably simple; single return type, no loops, no local variables and no side effects.

LAMBDA

Int sum = 10;

auto x = [sum](int d) mutable -> int {return sum += d; }

mutable, which is optional, means it changes a value.

If a lambda takes no arguments, the argument list can be omitted. [](){} becomes []{}

this keyword

Add *this* in the capture list to access class members.

Auto x = [this]() { // access class members }

CLASSES

When a class is responsible for an object accessed through a pointer, use a copy constructor and move assignment. A move constructor does not take a const argument.

After a move, a moved-from object should be in a state that allows a destructor to be run.

Using the default move or copy for a class in a hierarchy is typically a disaster. We simply don’t know what members the derived class has. The best thing to do is usually to delete the default copy and move.

Copy

ClassName(const ClassName&) = delete;

ClassName& operator=(const ClassName&) = delete;

If you need to copy an object in a class hierarchy, write some kind of clone function.

ARRAYS

Arrays cannot directly be passed by value. An array is passed as a pointer to its first element (pointer value):

Void comp(double arg[10])

Same as:

Void comp(double\* arg)

* Avoid multidimensional arrays, define suitable containers instead.
* Avoid passing arrays as pointers, as the size is not available. Passing the size as another argument to a function is a workaround. Instead, pass a container by reference.

POINTERS

\* = contents of

Use nullptr rather than 0 or NULL.

Applying *delete* to nullptr has no effect, so no need to check existence.

REFERENCES

& = address of

There are 3 kinds of references:

1. Lvalue: to refer to objects whose value we want to change
2. Const (lvalue): to refer to objects we don’t want to change
3. Rvalue: to refer to objects whose values we do not need to preserve after we have used it

Rvalue reference swap:

*Old style*

Void swap(T& a, T& b) {

T tmp {a} // now 2 copies of a

a = b; // now 2 copies of b

b = tmp; // now 2 copies of tmp

}

If T is a type for which it can be expensive to copy elements such as string or vector, this swap function becomes an expensive operation.

*Instead*

Void swap(T& a, T& b) {

T tmp {static\_cast<T&&>(a)};

a = static\_cast<T&&>(b);

b = static\_cast<T&&>(tmp);

}

With the std library you can use std::move() which does this for you, e.g.

T tmp{std::move(a)};

a = std::move(b);

Only use rvalue references for forwarding and move semantics

POINTERS VS REFERENCES

Use a pointer when:

* You need to change which object to refer to

Use a reference when:

* You want to be sure that a name always refers to the same object
* You want to use a user-defined (overloaded) operator on something that refers to an object

Pass by pointer vs by reference:

* A pointer can be reassigned, a reference cannot
* A pointer can be assigned null, reference cannot
* Pointers can iterate over an array
* A pointer uses -> to access members, references use .
* A pointer needs to be dereferenced with \* to access the memory location it points to, whereas a reference can be used directly.

ENUMERATIONS

Prefer enum classes:

Enum class Color {red, blue, green}

Enum class Traffic {green, yellow, red}

Color col = Color::red;

This makes the enum strongly typed and scoped. We cannot mix up traffic and color ‘red’.

EXCEPTIONS

Common std exceptions:

* out\_of\_range
* length\_error
* bad\_alloc

CONTAINERS

Prefer *auto* when working with iterators:

auto iBegin = vector.begin();

auto iEnd = vector.end();

std::algorithms

std::vector<int> v1 = {3, 2, 1, 5, 4};

std::sort(v1.begin(), b1.end());

std::binary\_search(v1.begin(), v1.end(), 4); // output 0 or 1

std::algorithms using predicates

template<typename T>

bool isEven(T n) { return n % 2 == 0; }

std::vector<int> v1 = {2, 4, 6, 20, 10};

std::all\_off(v1.begin(), v1.end(), isEven<int>); // true. Similar to js *every*

// also have any\_off, none\_off

auto it = find(v1.begin(), v1.end(), 20); // returns iterator. End if not found

if (it != v1.end()) {

cout << “Found at index: “ << it – v1.begin();

}

// also have find\_if (takes predicate as 3rd argument), find\_if\_not

auto c = count(v1.begin(), v1.end(), 2); // count occurrences of 2

// also have count\_if (takes predicate as 3rd argument)

replace(v1.begin(), v1.end(), 5, 99); // replace 5 with 99

// also have replace\_if (takes predicate as 3rd argument)

auto it = remove(v1.begin(), v1.end(), 42); // removes 42.

// returns iterator which points to the last element that was not removed (or the end if nothing removed).

// if element was removed, remember to resize the container

V1.resize(it – v1.begin());

Merge(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());

GENERAL

Assertions

Assert() // run time

Static\_assert() // compile time

If the NDEBUG macro is defined before the inclusion of <assert.h> then asserts are ignored. This is good for production code.

Using std::cin to populate array

Std::vector<int> read(std::istream& is) {

Std::vector<int> v;

For (double d : v) {

v.push\_back(d);

}

Return v;

}

Read(std::cin);

Std::initializer\_list

Used to define the initializer list constructor

Class List {

Public:

List(std::initializer\_list<int> ls) {

// loop through and push to vector perhaps?

}

}

List list = {10, 20, 30, 40, 50, 60}

Using IO for user defined types

Struct Entry {

String name;

Int number;

}

Ostream& operator<<(ostream& os, const Entry& e) {

Return os << e.name << ‘ ‘ << e.number;

}

Cout << entryObject;

Concurrency

Tasks are a computation that can potentially be executed concurrently with other computations.

Threads are a system-level representation of a task in a program.

#include <thread>

Std::thread t1 {f} // where f is a function

// executes in a separate thread

t1.join() // wait for t1 to finish before exiting program

Threads of a program share a single address space, process do not, this means threads can communicate with each other through shared objects.

Time

Time utilities are in the <chrono> header and std::chrono namespace.

auto t0 = high\_resolution\_clock::now();

do\_work();

auto t1 = high\_resolution\_clock::now();

cout << duration\_cast<milliseconds>(t1 – t0).count(); // cast to ms from ns

Type predicates

<type\_traits> header

Int num = 10;

is\_arithmetic<num>(); // true

others:

* is\_class()
* is\_pod()
* is\_literal\_type()
* is\_base\_of()
* etc…

Type aliases

using pchar = char\* // pointer to a char

Use an alias to define a meaningful name for a built-in type in cases which the built-in type used to represent a value might change.

Use enums and classes to define new types.

Const

When using a pointer, 2 objects are involved: the pointer itself, and the object pointed to.

*Easier to understand when read right to left*

Char\* const cp; // const pointer to char

Char const\* pc; // pointer to const char

Const char\* pc2; // pointer to const char

Functor

A functor is a class with an operator() overload

Template<typename T>

Class MultiplyBy {

Private:

T multipier mult;

Public:

MultiplyBy(T n) : mult(n) {}

T operator() (T n) { return mult \* n; }

}

MultiplyBy<int> x(9);

x(5); // 45

x(10); // 90

Generic

* Prefer scoped variables over free-store when possible
* Use pass by value for small objects
* Use pass by const reference to pass large values you don’t want to modify
* Use rvalue references to implement move and forwarding
* Use pass by reference only if you have to